A General and Parallel Platform for Mining Co-Movement Patterns over Large-scale Trajectories

ABSTRACT

1. INTRODUCTION

The prevalence of positioning devices has drastically boosted the scale and spectrum of trajectory collection to an unprecedented level. Tremendous amounts of trajectories, in the form of sequenced spatial-temporal records, are continually generated from animal telemetry chips, vehicle GPSs and wearable devices. Data analysis on large-scale trajectories benefits a wide range of applications and services, including traffic planning [1], animal analysis [2], and social recommendations [3], to name just a few.

A crucial task of data analysis on top of trajectories is to discover co-moving patterns. A *co-movement* pattern [4] refers to a group of objects traveling together for a certain period of time and the group is normally determined by spatial proximity. A pattern is prominent if the size of the group exceeds M and the length of the duration exceeds K, where M and K are parameters specified by users. Rooted from such basic definition and driven by different mining applications, there are a bunch of variants of co-movement patterns that have been developed with more advanced constraints.

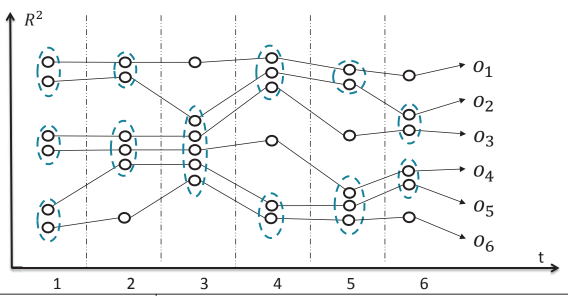
Table 1 summarizes several popular co-moving pattern s with different constraints in the attributes of clustering in spatial proximity, consecutiveness in temporal duration and computational complexity. In particular, the *flock* [5] and the *group* [6] patterns require all the objects in a group to be enclosed by a disk with radius r; whereas the *convoy* [7], the *swarm* [8] and the *platoon* [9] patterns resort to density-based spatial clustering. In the temporal dimension, the *flock* [5] and the *convoy* [7] require all the timestamps of each detected spatial group to be consecutive, which is referred to as *global consecutiveness*; whereas the *swarm* [8] does not impose any restriction. The *group* [6] and the *platoon* [9] adopt a compromised manner by allowing arbitrary gaps between the consecutive segments, which is called *local consecutiveness*. They introduce a parameter L to control the minimum length of each local consecutive segment.

	Proximity	Consecutiveness	Time Complexity
flock [10]	disk-based	global	$O(\lvert\mathbb{O}\rvert\lvert\mathbb{T}\rvert(M + log(\lvert\mathbb{O}\rvert))$
convoy [7]	density-based	global	$O(\lvert\mathbb{O}\rvert^2 + \lvert\mathbb{O}\rvert\lvert\mathbb{T}\rvert)$
swarm [8]	density-based	-	$O(2^{\lvert\mathbb{O}\rvert}\lvert\mathbb{O}\rvert\lvert\mathbb{T}\rvert)$
group [6]	disk-based	local	$O(\lvert\mathbb{O}\rvert^2\lvert\mathbb{T}\rvert)$
platoon [9]	density-based	local	$O(2^{\lvert\mathbb{O}\rvert}\lvert\mathbb{O}\rvert\lvert\mathbb{T}\rvert)$

Table 1: Constraints and complexity of co-movement patterns. The time complexity indicates the performance in the worst case, where $|\mathbb{O}|$ is the total number of objects and $|\mathbb{T}|$ is the number of descritized timestamps.

Figure 1 is an example to demonstrate the concepts of various co-movement patterns. The trajectory database consists of six moving objects and the temporal dimension is discretized into six snapshots. In each snapshot, we treat the clustering methods as a black-box and assume that they generate the same clusters. Objects in proximity are grouped in the dotted circles. As aforementioned, there are three parameters to determine the co-movement patterns and the default settings in this example are $M = 2$, $K = 3$ and $L = 2$. Both the *flock* and the *convoy* require the spatial clusters to last for at least K consecutive timestamps. Hence,$\{o_3, o_4\}$ and $\{o_5, o_6\}$ remains the only two candidates matching the patterns. The *swarm* relaxes the pattern matching by discarding the temporal consecutiveness constraint. Thus, it generates many more candidates than the *flock* and the *convoy*. The *group* and the *platoon* add another constraint on local consecutiveness to retain meaningful patterns. For instance, $\{o_1, o_2 : 1, 2, 4, 5\}$ is a pattern matching local consecutiveness because timestamps $\{1, 2\}$ and $\{4, 5\}$ are two segments with length no smaller than $L = 2$. The difference between the *group* and the *platoon* is that the *platoon* has an additional parameter K to specify the minimum number of snapshots for the spatial clusters. This explains why $\{o_3, o_4, o_5 : 2, 3\}$ is a *group* pattern but not a *platoon* pattern.

As can be seen, there are various co-movement patterns requested by different applications and it is cumbersome to design a tailored solution for each type. In addition, despite the generality of the *platoon* (i.e., it can be reduced to other types of patterns via proper parameter settings), it suffers from the so-called *loose-connection* anomaly. We use two objects o_1 and o_2 in Figure 2 as an example to illustrate the scenario. These two objects form a *platoon* pattern in timestamps $\{1, 2, 3, 102, 103, 104\}$. However, the two consecutive segments are 98 timestamps apart, resulting in a false positive co-movement pattern. In reality, such an anomaly may be caused by the periodic movements of unrelated ob-



Flock(M=2, K=3)	$\{o_3, o_4$: 1,2,3} $\{o_5, o_6$: 3,4,5}
Convoy(M=2, K=3)	$\{o_3, o_4$: 1,2,3} $\{o_5, o_6$: 3,4,5}
Group(M=2, L=2)	$\{o_1, o_2$: 1,2,4,5} $\{o_3, o_4, o_5$: 2,3} $\{o_5, o_6$: 3,4,5}
Platoon (M=2,K=3,L=2)	$\{o_1, o_2$: 1,2,4,5} $\{o_3, o_4$: 1,2,3} $\{o_4, o_5$: 2,3,5,6} $\{o_5, o_6$: 3,4,5}
Swarm (M=2,K=3)	$\{o_1, o_2$: 1,2,4,5} $\{o_3, o_4$: 1,2,3} $\{o_4, o_5$: 2,3,5,6} $\{o_5, o_6$: 1,3,4,5} $\{o_2, o_3$: 3,4,6}

Figure 1: Trajectories and co-movement patterns; The example consists of six trajectories across six snapshots. Objects in spatial clusters are enclosed by dotted circles. M is the minimum cluster cardinality; K denotes the minimum number of snapshots for the occurrence of a spatial cluster; and L denotes the minimum length for local consecutiveness.

jects, such as vehicles stopping at the same petrol station or animals pausing at the same water source. Unfortunately, none of the existing patterns have directly addressed this anomaly.

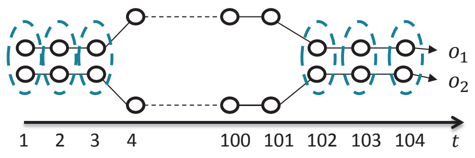
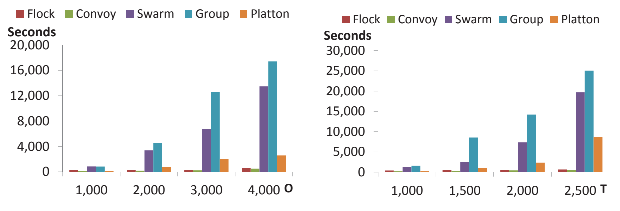


Figure 2: *Loose-connection* anomaly. Even though $\{o_1, o_2 : 1, 2, 3, 102, 103, 104\}$ is considered as a valid *platoon* pattern, it is highly probable that these two objects are not related as the two consecutive segments are 98 timestamps apart.

The other issue with existing methods is that they are built on top of centralized indexes which may not be scalable. Table 1 shows their theoretical complexities in the worst cases and the largest real dataset ever evaluated in previous studies is up to million-scale points collected from hundreds of moving objects. In practice, the dataset is of much higher scale and the scalability of existing methods is left unknown. Thus, we conduct an experimental evaluation with 4000 objects moving for 2500 timestamps to examine the scalability. Results in Figure 3 show that their performances degrade dramatically as the dataset scales up. For instance, the detection time of *swarm* drops five times as the number of objects grows from *1k* to *2k*. Similarly, the performance of *group* drops over seven times as the number of snapshots grows from *1.4k* to *2.4k*. These observations imply that existing methods are not scalable to support large-scale trajectory databases.

Therefore, our primary contributions in this paper are to close these two gaps. First, we propose the *general co-movement pattern* (GCMP) which models various co-moment patterns in a unified way and can avoid the *loose-connection* anomaly. In GCMP, we introduce a new gap parameter G to pose a constraint on the temporal gap between two consecutive segments. By setting a feasible G, the loose-connection anomaly can be avoided. In addition, our GCMP is both general and expressive. It can be reduced to any of the previous patterns by customizing the parameters.



(a) $1k$ objects (b) $1.2k$ timestamps

Figure 3: Performance measures on existing co-movement patterns. A sampled Geolife data set is used with up two 2.4 million data points.

Second, we investigate deploying our GCMP detector on MapReduce platforms (such as Hadoop and Spark) to tackle the scalability issue. Our technical contributions are three-fold. First, we replicate the snapshots in multiple data chunks to support efficient parallel processing. Second, we devise a novel *Star Partition and Mining* (SPM) algorithm as a fine-granularity partitioning strategy to achieve workload balance. For each star, an Apriori method is adopted to mine the co-movement patterns. Third, we propose two types of optimization techniques to further improve the performance, including *edge-simplification* to boost the shuffle process and *temporal monotonicity pruning* and *forward closure checking* to significantly reduce the number of enumerated candidates in the Aproiro algorithm.

We conduct a set of extensive experiments on XXX datasets with billion-scale trajectory points. The results show that XXX.

The rest of our paper is organized as follows: Section 2 summarizes the relevant literature on trajectory pattern mining. Section 3 states the problem definition of our general co-movement pattern mining. Section **??** provides an overview of our parallel architecture. Solutions to mining the general co-movement patterns are presented in Section **??** and Section 5. Section 6 discusses various optimization techniques to boost the system performance; Section 7 conducts extensive experiments to verify the usefulness and efficiency of our system. Finally Section 8 concludes the paper.

2. RELATED WORKS

The *co-movement patterns* in literature consist of five members, namely *group* [6], *flock* [10], *convoy* [7], *swarm* [8] and *platoon* [9]. We have demonstrated the semantics of these patterns in Table 1 and Figure 1. In this section, we focus on comparing the techniques used in these works. For more trajectory patterns other than *co-movement patterns*, interested readers may move to [11] for a comprehensive survey.

2.1 Flock and Convoy

The difference between *flock* and *convoy* lies in the object clustering methods. In *flock* objects are clustered based on their distance. Specifically, the objects in the same cluster needs to have a pair-wised distance less than *min_dist*. This essentially requires the objects to be within a disk-region of delimiter less than *min_dist*. In contrast, *convoy* cluster the objects using density-based clustering [12]. Technically, *flock* utilizes a m^{th}-order Voronoi diagram [13] to detect whether a subset of object with size greater than m stays in a disk-region. *Convoy* employs a trajectory simplification [14] technique to boost pairwise distance computations in the density-based clustering. After clustering, both *flock* and *convoy* use a line-sweep method to scan each snapshots. During the scan, the object group appears in consecutive timestamps is detected. Meanwhile, the object groups that do not match the consecutive constraint are pruned. However, such a method faces high complexity issues when supporting other patterns. For instance, in *swarm*, the candidate set during the line-sweep grows exponentially, and many candidates can only be pruned after the entire snapshots are scanned.

2.2 Group, Swarm and Platoon

Different from *flock* and *convoy*, all the *group*,*swarm* and *platoon* patterns have more constraints on the pattern duration. Therefore, their techniques of mining are of the same skeleton. The main idea of mining is to grow object set from an empty set in a depth-first manner. During the growth, various pruning techniques are provided to prune unnecessary branches. *Group* pattern uses the Apriori property among patterns to facilitate the pruning. *Swarm* adapts two more pruning rules called backward pruning and forward pruning. *Platoon* further adapts a prefix table structure to guide the depth-first search. As shown by Li et.al. [9], *platoon* outperforms other two methods in efficiency. However, the three patterns are not able to directly discover the general co-movement pattern. Furthermore, their pruning rules heavily rely on the depth-first search nature, which lost its efficiency in the parallel scenario.

2.3 MaReduce Framework

MapReduce (MR) was formally proposed by Dean et.al. [15] and has subsequently implemented by many open source systems. Those systems provide handy APIs with fault tolerances and are popularly used as large-scale data processing platforms. In simple words, there are two conceptual types of computing nodes in MR, namely the *mapper*s and the *reducer*s. The execution of a MR algorithm consists of three major steps: First, input data are partitioned and read by a *map* function on each mapper. Then, mappers emit key-value pairs which are *shuffle*d over the network to reducers. Lastly, reducers process the received data using a *reduce* function.

3. DEFINITIONS

Let $\mathbb{O} = \{o_1, o_2, ..., o_n\}$ be the set of objects and $\mathbb{T} = \{1, 2, ..., m\}$ be the descritized temporal dimension. A time sequence T is defined as a subset of $\mathbb{T}$, i.e., $T \subseteq \mathbb{T}$, and we use $|T|$ to denote sequence length. Let T_i be i-th entry in T and we say T is consecutive if $\forall 1 \leq i \leq |T|-1$, $T_{i+1} = T_i+1$. It is obvious that any time sequence T can be decomposed into consecutive segments and we say T is *L-consecutive* [9] if the length of all the consecutive segments is no smaller than L.

As illustrate in Figure 2, patterns adapting the notion of L-consecutiveness (e.g., *platoon* and *group*) still suffer from *loose connection* problem. To avoid such an anomaly without losing pattern generality, we introduce a parameter G to control the gaps between timestamps in a pattern. Formally, a G-connected time sequence is defined as follows:

Pattern	M	K	L	G	Clustering
Group	2	1	2	$\|T\|$	Disk-based
Flock	·	·	K	1	Disk-based
Convoy	·	·	K	1	Density-based
Swarm	·	·	1	$\|T\|$	Density-based
Platoon	·	·	·	$\|T\|$	Density-based

Table 2: Expressing other patterns using GCMP. · indicate a user specified value. M represents the object *size* constraints. K represents *duration* constraint. L represents *consecutiveness* constraint. G represents the *connection* constraints.

Definition 1 (*G*-connected). *A time sequence T is G-connected if the gap between any of its neighboring timestamps is no greater than G. That is $\forall T_i, T_{i+1} \in T, T_{i+1} - T_i \leq G$.*

We take $T = \{1, 2, 3, 5, 6\}$ as an example, which can be decomposed into two consecutive segments $\{1, 2, 3\}$ and $\{5, 6\}$. T is not 3-consecutive since the length $\{5, 6\}$ is 2. Thus, it is safe to say either T is 1-consecutive or 2-consecutive. On the other hand, T is 2-connected since the maximum gap between its neighboring time stamps is $5 - 3 = 2$. It is worth noting that T is not 1-connected because the gap between T_3 and T_4 is 2 (i.e., 5-3=2).

Given a trajectory database descritized into snapshots, we can conduct a clustering method, either disk-based or density-based, to identify groups with spatial proximity. Let T be the set of timestamps in which a group of objects O are clustered. We are ready to define a more general co-movement pattern:

Definition 2 (General Co-Movement Pattern). *A general co-movement pattern finds a set of objects O satisfying the following five constraints: 1) closeness: the objects in O belong to the same cluster in the timestamps of T; 2) significance: $|O| \geq M$; 3) duration: $|T| \geq K$; 4) consecutiveness: T is L-consecutive; and 5) connection: T is G-connected.*

There are four parameters in our general co-movement pattern, including object constraint M and temporal constraints K, L, G. By customizing these parameters, our pattern can express other patterns proposed in previous literature, as illustrated in Table 2. In particular, by setting $G = |T|$, we achieve the *platoon* pattern. By setting $G = |T|, L = 1$, we achieve the *swarm* pattern. By setting $G = |T|$, $M = 2$, $K = 1$, we gain the *group* pattern. Finally by setting $G = 1$, we achieve the *convoy* and *flock* pattern. In addition to the flexibility of representing other existing patterns, our GCMP is able to avoid the *loose connection* anomaly by tuning the parameter G. It is notable that GCMP cannot be modeled by existing patterns.

It is also observable that the number of patterns in GCMP could be exponential under some parameter settings (i.e., when expressing *swarm*). In particular, given a parameter M, if a pattern P is valid, then any subset of P with size M is also a valid pattern. This results in additional $\Sigma_{M \geq i \geq |P.O|}\binom{|P.O|}{i}$ patterns, which is clearly overwhelming and redundant. For all these patterns, output P is sufficient.

Therefore, we define the *Closed General Co-Movement Pattern* as follows:

Definition 3 (Closed General Co-Movement Pattern)**.** *A general co-moving pattern $P = \langle O : T \rangle$ is closed if and only if there does not exist another general co-moving pattern P' s.t. $P.O \subseteq P'.O$.*

For example, let $M = 2, K = 2, , L = 1, G = 1$. In Figure 1, the pattern $P_1 = \{o_3, o_4 : 1, 2, 3\}$ is not a closed pattern. This is because $P_2 = \{o_3, o_4, o_5 : 2, 3\}$ is a closed pattern since $P_2.O \supset P_1.O$. The closed pattern avoids outputting duplicate information, thus making the result patterns more compact.

Our definition of GCMP is free from clustering method. Users are able to supply different clustering methods to facilitate different application needs. We currently expose both disk-region based clustering and DBSCAN as options to the user.

In summary, the goal of this paper is to present a parallel solution for discovering closed GCMP from large-scale trajectory data.

Before we move on to the algorithmic part, we list the notations that are used in the following sections.

Symbols	Meanings
Tr_i	Trajectory of object i
S_t	Snapshot of objects at time t
$\mathbb{O}$	Set of objects
M	Object size constraint
K	Duration constraint
L	Consecutiveness constraint
G	Connection constraint
T	Time sequence
$C_t(o)$	the cluster of object o at time t
Sr_i	The star structure of object i

Table 3: Notions that will be used

4. TEMPORAL REPLICATION AND MINING

We resort to the MapReduce (MR) paradigm for designing a parallel solution in mining GCMP. It is straightforward to partition the trajectory database into equal-sized temporal segments; and then mining the GCMP patterns out of each segments. However, GCMP patterns may cross multiple temporal segments. In order to ensure the correctness of results, we need to guarantee that every valid patterns can be mined within at least one partitions. Thus, some snapshots need to be replicated several times in multiple partitions. This leads us to design the *Temporal Replication and Mining* (TRM) algorithm.

The work flow of TRM is illustrate as in Figure 4. In brief, there are two pipeline MR jobs which further consist of four stages. The first MR job is considered as a preprocessing, where input trajectories are grouped into snapshots and each snapshot runs a clustering method (e.g., Disk-based, DBSCAN, etc.). The output is shown as in Figure 4(b). The second MR job is the *TRM* algorithm. In the *map* phase (i.e., Figure 4 (c)), temporally closed snapshots are grouped into a partition. In the *reduce* phase (i.e., Figure 4 (d)), a *Line Sweeping* method is developed to discover GCMP in each partition. Finally, patterns from different partitions are then collected to form the output.

Since in the first MR job, each partition contains only one snapshot for clustering, it is not necessary to replicate any snapshot. Thus, we focus on describing the second MR job which is the *Temporal Replication and Mining* algorithm. We use R to denote the replication factor. The outline of TRM is shown in Algorithm 1.

Algorithm 1 Temporal Replication and Mining

```
Require: list of $\langle t, S_t \rangle$ pairs
 1: $R \leftarrow (\lceil \frac{K}{L} \rceil - 1) * G + 2K$
 2: —Map Phase—
 3: for all $\langle t, S_t \rangle$ do
 4:     for all $i \in 1...R$ do
 5:         emit a $\langle t - i, S_t \rangle$ pair
 6:     end for
 7: end for
 8: —Partition and Shuffle Phase—
 9: for all $\langle t, S \rangle$ pair do
10:     group-by $t$, emit a $\langle t, Par_t \rangle$
11:     where $Par_t = \{S_t, S_{t+1}, ..S_{t+R}\}$
12: end for
13: —Reduce Phase—
14: for all $\langle t, Par_t \rangle$ do
15:     lineSweepMining($Par_t$)
16: end for
```

As shown in Algorithm 1, the TRM algorithm contains three steps. First, in the map phase, each snapshot is keyed with its timestamp (lines 2-7). Second, in the partition phase, every snapshot is grouped with its next R snapshots to form a partition (lines 8-12). We will shortly discuss how the R value is derived. Third, in the reduce phase, a line sweeping method is invoked to mine GCMP within each partition (lines 13-15). It is easy to see that this method replicates a snapshots at most R times. The replication factor R is critical for the performance of TRM. If the R is large, the shuffle cost as well as the reduce cost would be high. On the contrary, if R is small, valid patterns may be missed out. In the Algorithm 1, the R is chosen as $(\lceil \frac{K}{L} \rceil - 1) * G + 2K$. We will show later the correctness of this value.

4.1 Line Sweep Mining

Each task in the reduce phase processes a partition Par_i, which contains R snapshots starting from snapshot S_i. We observe that within each Par_i, only the patterns whose object sets are contained in the first snapshot are necessary to be reported. Therefore, we design a simple *line-sweep mining*(LSM) method for discovering GCMPs. The algorithm works as in Algorithm 2.

The algorithm scans snapshots in a partition in sequence. During the scan, it maintains a candidate set C which contains potential patterns (line 1). The algorithm starts by inserting clusters at S_i to C (lines 2-4). Subsequently, in each iteration, clusters in C are joined with clusters at S_i to generate a new set of patterns N(lines 6). The valid new patterns form a new candidate set C and any invalid patterns are discarded(lines 8 and 11).

4.2 Correctness of TRM

We prove the correctness of TRM from two aspects. First, the choice of $R = (\lceil \frac{K}{L} \rceil - 1) * G + 2K$ would not miss out

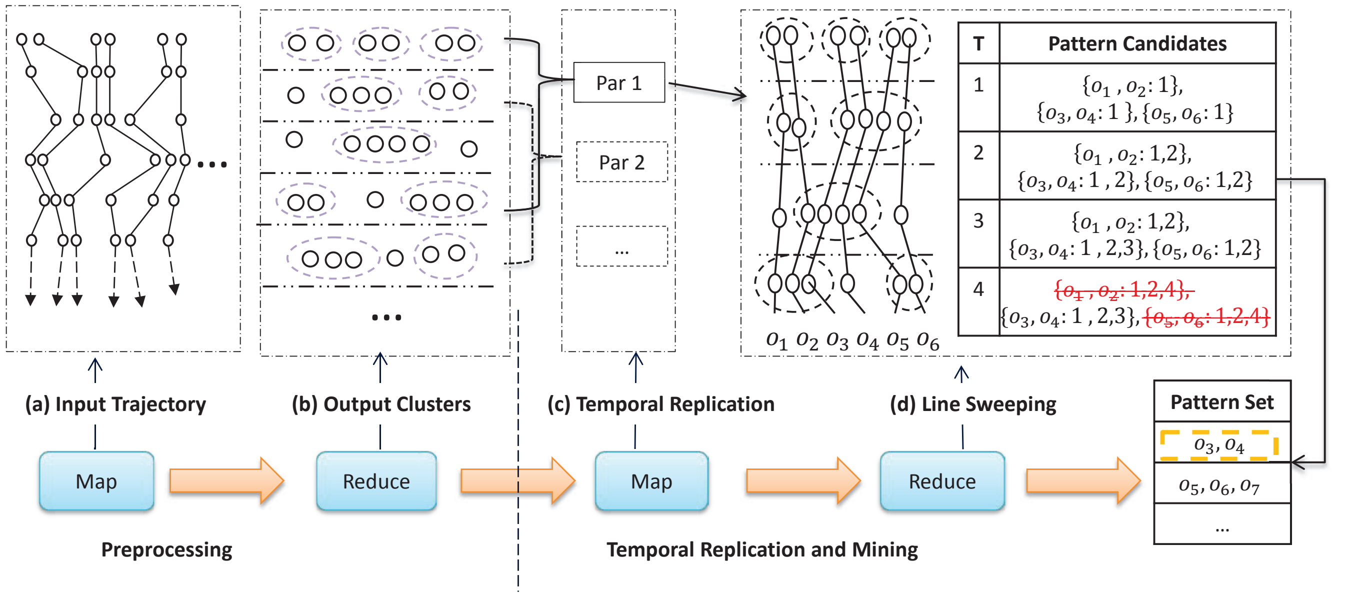

Par 1
Par 2
...
T
Pattern Candidates
1
$\{o_1, o_2: 1\}, \{o_3, o_4: 1\}, \{o_5, o_6: 1\}$
2
$\{o_1, o_2: 1,2\}, \{o_3, o_4: 1,2\}, \{o_5, o_6: 1,2\}$
3
$\{o_1, o_2: 1,2\}, \{o_3, o_4: 1,2,3\}, \{o_5, o_6: 1,2\}$
4
~~$\{o_1, o_2: 1,2,4\}$~~, $\{o_3, o_4: 1,2,3\}$, ~~$\{o_5, o_6: 1,2,4\}$~~
$o_1\ o_2\ o_3\ o_4\ o_5\ o_6$
(a) Input Trajectory
(b) Output Clusters
(c) Temporal Replication
(d) Line Sweeping
Map
Reduce
Map
Reduce
Pattern Set
o_3, o_4
o_5, o_6, o_7
...
Preprocessing
Temporal Replication and Mining

Figure 4: Work flow of Temporal Replication and Mining. (a)(b) correspond to the first MR job which computes the clusters at each snapshot; (c)(d) correspond to the second MR job which uses TRM to mine GCMP in parallel.

Algorithm 2 Line Sweep Mining

```
Require: $Par_t = \{S_t, S_{t+1}, ...\}$
 1: $C \leftarrow \{\}$                                   ▷ Candidate set
 2: for $c \in S_t$ do
 3:     $C$.add($\langle c, t \rangle$)
 4: end for
 5: for all $i \in [1, R]$ do
 6:     $N \leftarrow S_{t+i} \oplus C$
 7:     for all $n \in N$ do
 8:         if $|n.O| \geq M$ then $C$.add($n$).
 9:         end if
10:     end for
11:     remove unqualified candidate from $C$
12: end for
13: output qualified candidate in $C$
```

any valid patterns. Second, no false patterns are reported in any partitions. We formalize these two properties as *completeness* and *soundness* as follows:

Definition 4 (Completeness and Soundness)**.** *Let a partition method $\mathbb{P}$ partitions a trajectory database Tr into segments, $Par_1, ..., Par_m$. $\mathbb{P}$ is complete if for every valid pattern P in Tr, $\exists Par_i$ s.t. P is valid in Par_i. $\mathbb{P}$ is sound if for all patterns that are valid in any Par_i, they are also valid in TR.*

Apparently, in TRM, replicating the entire trajectories (i.e., $R = |\mathbb{T}|$) meets the *soundness* and *completeness* requirements. However, it burdens the network shuffle and limits the parallelism. We carefully chose $R = (\lceil \frac{K}{L} \rceil - 1) * G + 2K$ and use the following theorem to state the correctness:

Theorem 1 (Correctness of Replication)**.** *Temporal replication partition is sound and complete.*

Proof. The soundness of partition can be observed from the fact that each partition represents a consecutive segments of trajectories. Therefore patterns in a partition can be directly mapped back to original trajectories. For completeness, with a valid pattern P, let T' be the subsequence of $P.T$ which conforms to K, L, G with the smallest length. Note that there could be many qualified T's. Let the i^{th} local-consecutive segment of T' be l_i and let the i^{th} gap of T' be g_i. Then, the size of T' can be written as $\Sigma_i(l_i + g_i)$. Since T' conforms to K, L, G, then $2K \geq \Sigma_i(l_i) \geq K, l_i \geq L$, $g_i \leq G$. It follows: $\Sigma_i(l_i + g_i) \leq (\lceil \frac{K}{L} \rceil - 1) * G + 2K$. If every partition is of at least such a size, then T' must be captured by at least one of the partition. Thus, the pattern P would be valid in that partition. This proves the completeness. □

Example 1. *We illustrate the entire TRM method using Figure 4 (c)(d) with $M = 2, K = 2, L = 2, G = 2$. In Figure 4 (c), snapshots are combined into partitions with sizes equal to $(\lceil \frac{K}{L} \rceil - 1) * G + 2K = 4$. Then a line sweep method is performed in (d) for partition 1. Each C_i refers to the candidate set during sweeping snapshot i. Initially, C_1 contains patterns whose object set is in snapshot 1. As line sweeps, at snapshot 4, since the timestamps of $\{o_1, o_2\}$ and $\{o_5, o_6\}$ are both $\{1, 2, 4\}$ which violate the G constraint, thus the two candidates are removed from C_4. After all snapshots are swept, $\{o_3, o_4\}$ is the qualified pattern and is outputted.*

The TRM approach though achieves good parallelism, it requires to replicate the data multiple times. Specifically, each snapshots are replicate $(\lceil \frac{K}{L} \rceil - 1) * G + 2K$ times. In the cases of *swarm*, *group* and *platoon*, G is as large as $|T|$. Handling those cases is equivalent to replicate the entire snapshots to each partition, which surrenders the benefit of parallelism.

5. STAR PARTITION AND MINING

In order to achieve a good parallelism under any pattern parameters, we propose the *Star Partition and Mining*

(SPM) method. In SPM, we first design a novel object-based partition method named *star partition*. Then, we adapt an *Apriori* method to mine GCMP patterns out of each partition.

The overview of the SPM method is presented in Algorithm 3. As shown, the SPM method takes three phases. In the map phase, objects from the same cluster forms object-object pairs. The object-object pairs are then paired up with the timestamp of the snapshot to form a triplet(lines 1-8). In the partition phase, triplets with the same leading object form a *star* which will be explained shortly (lines 9-12). Lastly in the reduce phase, patterns are mined from each star structure (lines 13-16).

Algorithm 3 Star Partition and Mining

```
Require: list of $\langle t, S_t \rangle$ pairs
 1: —Map phase—
 2: for all $C \in S_t$ do
 3:     for all $(o_1, o_2) \in C \times C$ do
 4:         if $o_1 < o_2$ then
 5:             emit a $\langle o_1, o_2, \{t\} \rangle$ triplet
 6:         end if
 7:     end for
 8: end for
 9: —Partition and Shuffle phase—
10: for all $\langle o_1, o_2, \{t\} \rangle$ triplets do
11:     group-by $o_1$, emit $\langle o_1, Sr_{o_1} \rangle$
12: end for
13: —Reduce phase—
14: for all $\langle o, Sr_o \rangle$ do
15:     Apriori($Sr_o$)
16: end for
```

5.1 Star Partition

The intuition of the star partition is that, if two objects are part of the same pattern, they must belong to the same cluster at some snapshots. Therefore, we may link objects that belong to the same cluster to form a *connection graph*. Objects that are not connected surely fail to form a pattern. We may then partition the connection graph based on vertex connectivity s.t. mining GCMPs can be done in parallel. We define the *connection graph* and *star* as follows:

Definition 5 (Connection Graph and Star)**.** *A connection graph is an undirected graph $G = (V : E)$, where each $v \in V$ represents an object. An edge $e(s, t) = ET \in E$ contains all the timestamps at which s, t are in the same cluster, i.e., $\forall t \in ET, C_t(s) = C_t(t)$. A star of a vertex s, denoted as Sr_s, is the set of incidental edges on s whose another ending vertex is greater than s. I.e, $\forall e(s, t) \in Sr_s$, $s < t$. We name s as the* central vertex *of Sr_s.*

It is notable that we require vertices in a star to be greater than its central vertex. This effectively avoids replicating edges. *Connection graph* and *star* examples are shown in Figure 5 (a) and (b). In (a), a connection graph is formed based on the example in Figure 1. In (b), 5 stars are presented. It is easy to see that, by requiring the center vertex to be the smallest vertex in a star, there are no edges been replicated. In implementation, as stated in Algorithm 3 line 4, the comparison between vertices/objects are based on the vertex/object ID.

Although star partition is performed based on the object connections, each star can be effectively viewed as a subset of trajectories. To see this, each vertex in a star can be viewed as an object. The timestamps of center vertex s is the union of all the edges in Sr_s. The timestamps of vertex $v \neq s$ is the edge $e(s, v)$. Therefore, we are able to define and mine GCMP on the stars. Before describing the mining strategy, we first state in the following theorem that the star-partition is complete and sound:

Theorem 2 (Soundness and Completeness of Star Partition)**.** *Star partition is sound and complete.*

Proof. For the soundness, if P is a valid pattern in Sr_s, then at every time t, $\forall o_1, o_2 \in P.O$, $C_t(o_1) = C_t(o_2)$. By definition, P is valid in the original trajectories. For the completeness, if P is a valid pattern in original trajectories, let s be the object with smallest ID in $P.O$. Then by the definition of pattern, $\forall t \in P.T$, $\forall o \in P.O$, $C_t(s) = C_t(o)$. It follows that all object $o \in P.O$ are in Sr_s. Furthermore, every timestamp in $P.T$ is included in Sr_s. Therefore, P is a valid pattern in Sr_s. $\square$

Based on the above theorem, we can mine GCMP from each partition independently. It is notable that, in star partition, original data is replicated for $O(|\mathbb{O}|)$ times as each object may be sent to $O(|\mathbb{O}|)$ partitions. Since this complexity is free from pattern parameters, the star partition is more scalable than the temporal replication. In later sections, we will describe several engineering level optimization to further reduce the amount of replicated data.

5.2 Apriori Mining

In the mining phase, we need to find the patterns within each star. To systematically discover the patterns, we design the *Apriori Mining* method which is similar to the technique in frequent item mining literature. During the algorithm, we call a candidate pattern R-pattern if the size of its object set is R. Our algorithm runs in iterations. During each iteration R, we try to generate all $(R+1)$-patterns. In iteration 1, the 2-pattern is the edges in Sr_s. In particular, for each $e(s, v) = ET$, pattern $p = (\{s, v\}, ET)$ is formed. During each iteration, we generate $(R+1)$-patterns by joining R-patterns with 2-patterns. Specifically, the join between $p_1 = (O_1 : T_1)$ and $p_2 = (O_2 : T_2)$ would generate a new pattern $p_3 = (O_1 \cup O_2 : T_1 \cap T_2)$. Notice that in Sr_s, each R-pattern consists of the object s, thus the join will grow a R-pattern at most to a $(R+1)$-pattern. Our mining algorithm stops where no further patterns are generated. The algorithm is illustrated as in Algorithm 4.

An illustration of Algorithm 4 is shown in Figure 5 (c). As shown, the star $Sr_3 = \{3, 4, 5, 6\}$ initially generate three 2-candidates. At every iteration, higher level candidates are generated by joining lower level candidates. When no more candidates can be generated, the algorithm stops by outputting the valid patterns.

It is notable that Algorithm 4 takes exponential complexity to mine GCMP. There are two major factors dragging down the performance. First, the size of Sr_s affects the initial size of 2-patterns. Second, the candidates generated in each level affects the join performance. In later sections, we exploit the property of GCMP to reduce the two factors.

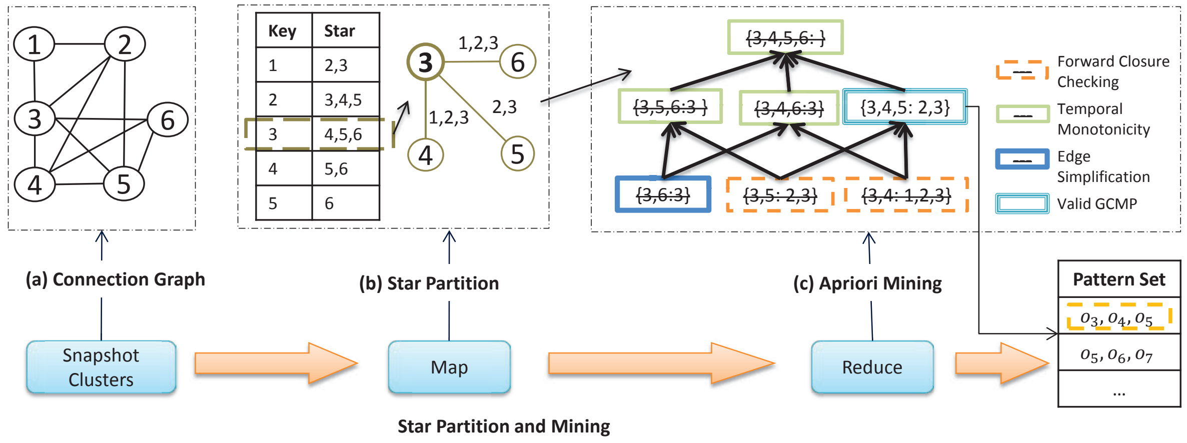


Figure 5: Star partition and mining. (a) Conceptual connection graph from Figure 1.(b) Five star partitions are generated (c) Apriori Mining with various pruning techniques.

Algorithm 4 Apriori Mining

```
Require: $Sr_s$
 1: Lv ← {},Ground ← {}, Output ← {}
 2: for all $e(s,t) = T \in Sr_s$ do
 3:     Ground.add($\langle\{s,t\},T\rangle$);
 4:     Lv ← Ground;
 5: end for
 6: while true do
 7:     if Lv is not empty then
 8:         LvCand ← {}
 9:         for all $cand_v \in Lv$ do
10:             for all $cand$ ∈Ground do
11:                 $p \leftarrow cand_v$ join $cand$
12:                 if $p.T$ is a candidate sequence then
13:                     LvCand.add($p$)
14:                 end if
15:             end for
16:         end for
17:         if $Lv$ is a pattern then
18:             Output.add($Lv$)
19:             break
20:         end if
21:         Lv ← LvCand
22:     else
23:         break
24:     end if
25: end while
26: output.addAll($Lv$)
27: return output
```

6. OPTIMIZATION

In this section, we describe several optimizations to the star-partition and mining algorithm. In addition, we also address some practical issues when deploying the SPM algorithm to real MapReduce based systems.

6.1 Edge Simplification

Each edge $e(s,v)$ in Sr_s contains a time sequence ET which represents the co-occurrence of s and v. We notice that the edge between s and t is not always necessary. For example, if an edge has a cardinality less than K, it is unnecessary to include this edge to Sr_s since it cannot contribute to any patterns. This motivates us to simplify the edges in Sr_s to boost the overall performance.

Our goal of edge simplification is to, given a time sequence T, find a subsequence of $T' \subseteq T$, such that T' is potentially conforms to K, L, G. And we wish $|T'|$ to be as small as possible. We star-off by observing that for every time sequence T, T can be divided into a set of maximally G-connected subsequences. Note that a maximally G-connected subsequence can potentially contribute to a pattern if it conforms to K, L. Therefore, we are able to reduce T to its maximally G-connected subseuqnces which conform to K, L.

To formally describe the idea, we define the *candidate sequence* as follows:

Definition 6 (Candidate Sequence)**.** *Given the pattern parameters: L, K, G, a sequence T is a* Candidate Sequence *if for any of its maximal G-connected sequence T', T' conforms to L, K.*

For example, let $L = 2, K = 4, G = 2$, sequence $T_1 = (1, 2, 4, 5, 6, 9, 10, 11, 13)$ is not a fully candidate sequence since one of its maximal G-connected sequence $(9, 10, 11)$ is not a partly candidate sequence. In contrast, sequence $T_2 = (1, 2, 4, 5, 6)$ is a fully candidate sequence.

To reduce a sequence T to a candidate sequence, we need to strip out its maximal G-connected subsequences which does not form to K, L. Such a reduction takes two rounds scan of T as shown in Algorithm 5. In the first round, the consecutive portions of T with size less than L are removed. In the second round, the maximal G-connected sequences of size less than K are removed. Clearly the simplification algorithm runs in linear time.

Algorithm 5 Edge Simplification

```
Require: $T$
 1: —Remove the consecutive segment with size less than $L$—
 2: $c \leftarrow 0$
 3: for $i \in (0, ..., |T|)$ do
 4:     if $T[i] - T[i-1]! = 1$ then
 5:         if $i - c < L$ then
 6:             $T$ remove $[c : i)$
 7:         end if
 8:         $c \leftarrow i$
 9:     end if
10: end for
11: —Remove the $G$-connected segment with size less than $K$—
12: $s \leftarrow 1$, $c \leftarrow 0$
13: for $i \in (0 : |T|)$ do
14:     if $T[i] - T[i-1] > G$ then
15:         if $s < K$ then
16:             $T$ remove $[c : i)$
17:         end if
18:         $c \leftarrow i$, $s \leftarrow 1$
19:     else
20:         $s{+}{+}$
21:     end if
22: end for
```

Example 2. *Take $T_1 = \{1, 2, 4, 5, 6, 9, 10, 11, 13\}$ as an example of edge simplification. Let $L = 2, K = 4, G = 2$. In the first round of scan. T_1 reduces to $\{1, 2, 4, 5, 6, 9, 10, 11\}$. The consecutive subsequence $\{13\}$ is removed by $L = 2$. T_1 has two maximal G-consecutive subsequences, which are $\{1, 2, 4, 5, 6\}$ and $\{9, 10, 11\}$. Since $K = 4$, $\{9, 10, 11\}$ is removed from T_1 in the second round of scan. Therefore, T_1 is simplified to $\{1, 2, 4, 5, 6\}$.*

By leveraging the edge simplification technique, the size of the edges in Sr_s can be greatly reduced. If an edge cannot be reduced to a candidate sequence, then it is directly removed from Sr_s. If an edge can be reduced to a candidate sequence, replacing itself by the candidate sequence results in a more compact storage.

6.2 Candidate Pruning

6.2.1 Temporal monotonicity

During the apriori phase, we repeatedly join candidate patterns in different levels to generate a larger set of a patterns. We observe that traditional monotonic property of Apriori algorithms **does not** hold in GCMP mining. That is given two candidate P_1, P_2, if $P_1.O \subset P_2.O$ and P_1 is not a valid pattern, then P_2 may or may not be a valid pattern. However, we notice that we may form another monotonic property based on the *candidate sequence* such that the Apriori algorithm could still benefit.

The intuition is that if a candidate $P_1.T$ cannot be reduce to a *candidate sequence*, then P_1 cannot be valid pattern. Furthermore, any candidate P_2, with $P_1.O \subset P_2.O$ cannot be a valid pattern. This *temporal monotonic property* is explicitly described as in the follow theorem:

Theorem 3 (Temporal Monotonic Property of GCMP)**.** *Given the temporal parameters L, G, K, for a candidate c in Algorithm 4, if $c.T$ cannot be reduced to a candidate sequence, then for any candidate c' with $c.O \subset c'.O$, c' can be pruned.*

Proof. Let c_1, c_2 be two candidates with $c_1.O \subset c_2.O$. It is easy to see that $c_1.T \supseteq c_2.T$. If $c_1.T$ cannot be reduced to a candidate sequence, then any subset of $c_1.T$ cannot be reduced. It follows that $c_2.T$ cannot be reduced neither. Thus, if $c_1.T$ cannot be reduced to a candidate sequence, c_2 can be pruned. □

6.2.2 Forward closure checking

Although leveraging *temporal monotonicity* could largely prune false candidates and reduce the apriori search space, it is ineffective when a *true* pattern exists. For example, if a final pattern of a star Sr_s is the *union of all vertices* in the star, then in apriori, $\binom{|Sr_s|}{i+1}$ candidates needs to be generated at each level i. This results in an exponential search space while the output only contains one pattern. In general, when candidates at level i collectively forms a true pattern, running aprior produces many wasted candidates.

Let Lv_i be the set of candidates at level i of Algorithm 4, we use the *forward closure FC_i* to denote the union of the objects in all candidates in Lv_i. Then, the *forward closure checking* is stated as follows:

Theorem 4 (Forward Closure Checking Rule)**.** *Let Lv_i be the candidates generated at level i in Algorithm 4, if FC_i is a proper pattern, then it is safe to terminate Algorithm 4 and directly output FC_i.*

Proof. We prove by contradiction. Suppose there exists another pattern P such that $P.O \neq FC_i$, let $X = P.O - FC_i \neq \emptyset$. Consider a subset of P which contains X with size $i+1$, (i.e., $P_1 \subseteq P, P_1 \subseteq X, |P_1| = i+1$). Since P is a proper pattern, then P_1 is also a proper pattern. Therefore $P_1 \in Lv_i$. Then it follows X is in the forward closure of FC_i, (i.e.,$X \in FC_i$), which contradicts with $X \notin FC_i$. □

It is notable that, as the level grows in Algorithm 4, the closure FC reduces, thus the pruning power of FC would be stronger.

Example 3. *We use Figure 5 (c) to demonstrate the power of candidate pruning. As shown, at the initial stage, $\{3, 6 : 3\}$ is first pruned by Edge Simplification since its timestamps fails to be a candidate sequence. Subsequently, all further candidates containing $\{3, 6\}$ are pruned by Temporal Monotonicity. Then, we check the Forward Closure of remaining candidates (i.e., $\{3, 4\}$ and $\{3, 5\}$) and find $\{3, 4, 5\}$ is a valid candidate. Therefore, $\{3, 4\}$ and $\{3, 5\}$ are pruned, and $\{3, 4, 5\}$ is the output.*

6.3 Star Partition Analysis

As we see in Section 5, the apriori phase in SPM may take exponential time. An important factor affecting the performance of SPM is the size of stars. We notice that, the size of star is related to the way the vertexes are numbered.

For example, Figure 6 gives an alternative numbering of vertexes in connection graph thus produces a different set of stars. This partitioning constructs four stars with the maximum star consisting of 5 edges. Compared to the partitioning in Figure 5(b) where five stars are produced with maximum star of 3 edges, this partition is inferior in two aspects. First, this partition lacks of one star, which results

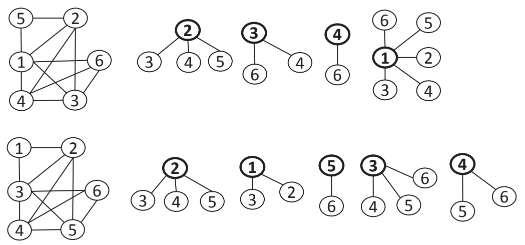


Figure 6: An alternative numbering and partitioning of the connection graph in Figure 5.

in a smaller number of parallelism. Second, this partition has a larger size of maximum edges. With the exponential time complexity in Apriori, the largest star takes much more time to compute.

The example shows that different numbering of vertexes in connection graph indeed affects the SPM performance. Therefore we wish to find a numbering scheme of connection graph such that the edges in each star are best distributed. Notice that the sum of edges is invariant for any numbering scheme of the graph. Thus, our goal is equivalent to find a numbering such that the maximum edge size from all stars is minimized.

To formalize the objective, we design an linear algebra model as follows: Let G denote a connection graph. Let $\mathbb{A}$ be an arbitrary numbering of vertexes in G. Let $(A : a_{i,j})$ be the matrix representing the induced graph wrt. $\mathbb{A}$. Let vector $\vec{b}$ be the *one*[1] vector. Let $\vec{c} : c_j$ be the vector equals to $A\vec{b}$. It is easy to see that each c_j denotes the size of star for vertex j. Therefore, the objective of load balancing can be formalized as follows:

$$\mathbb{A} = \text{argmin}(||A\vec{b}||_\infty),\text{where } ||A\vec{b}||_\infty = \max_{1\le j\le n}(c_j) \quad (1)$$

Though the objective is well-defined as above, it is challenging to directly optimize the equation. First, suppose there are n vertex in G, enumerating all possible $\mathbb{A}$s leads to $n!$ combinations. Such a high complexity is trivially unpractical. Second, since G is constructed at runtime, the load planning can only start after G is created. The planning time are thus required to be short enough otherwise the benefit cannot payoff the planning time.

Despite these challenges, we observe that there is a $O(1)$ time solution which is good enough as stated in the following theorem.

Theorem 5 (Balance of Star Partition)**.** *Let G be a connection graph with n vertexes and the average degree d. Let $\mathbb{A}^*$ be the optimal numbering wrt. Equation 1. For any numbering, $\mathbb{A}$, with high probability, the absolute difference between $\mathbb{A}^*$ and $\mathbb{A}$ is $O(\sqrt{n\log n})$. That is, it is very likely that $||\mathbb{A}\vec{b}||_\infty = ||\mathbb{A}^*\vec{b}||_\infty + O(\sqrt{n\log n})$.*

Proof. Let $\mathbb{A}^*$ be the optimal solution wrt Equation 1. Since we have a star for each object, by the degree-sum formula and pigeon-hole theorem, $||A^*\vec{b}||_\infty \ge d/2$. Next, let $e_{i,j}$ be a 0-1 indicator variable determining whether vertex i connects vertex j in G. Note that edges in G are independent. We use d_i to denote the degree of object i in G. It follows that $E[d_i] = E[\Sigma_{1\le j\le n} e_{i,j}] = d$. Since in the star partition, each edge is assigned to the ending vertex with lower ID, the connection between $a_{i,j}$ and $e_{i,j}$ can be written as:

$$a_{i,j} = \begin{cases} e_{i,j}, i > j \\ 0, otherwise \end{cases}$$

There are two observations made on the above equation. First, since $e_{i,j}$s are independent, $a_{i,j}$s are independent. Second, since $i > j$ and $e_{i,j}$ are independent. $E[a_{i,j}] = E[e_{i,j}]E[i > j] = E[e_{i,j}]/2$.

By definition, $c_i = \Sigma_{1\le j\le n} a_{i,j}$, is a sum of n independent 0-1 variables. Taking expectation on both sides, we get: $E[c_i] = E[\Sigma_{1\le j\le n} a_{i,j}] = E[\Sigma_{1\le j\le n} e_{i,j}]/2 = d/2$. Let $\mu = E[c_i] = d/2$, $t = \sqrt{n\log n}$, by Hoeffding's Inequality, the following holds:

$$\begin{aligned} Pr(c_i \ge \mu + t) &\le \exp(\frac{-2t^2}{n}) \\ &= \exp(-2\log n) = n^{-2} \end{aligned}$$

The first step is due to the fact that all $a_{i,j}$ are bounded in the range of [0,1]. Next, since the event $(\max_{1\le j\le n}(c_j) \ge \mu + t)$ can be viewed as $\cup_{c_i}(c_i \ge \mu + t)$, by Union Bound, we achieve the following:

$$\begin{aligned} Pr(||A\vec{b}||_\infty \ge \mu + t) &= Pr(\max_{1\le j\le n}(c_j) \ge \mu + t) \\ &= Pr(\cup_{c_i}(c_i \ge \mu + t)) \\ &\le \Sigma_{1\le i\le n} Pr(c_i \ge \mu + t) \\ &= n^{-1} = 1/n \end{aligned}$$

Substitute t and μ, we achieve the following concise form:

$$Pr(||A\vec{b}||_\infty \ge (d/2 + \sqrt{n\log n})) \le 1/n$$

This indicates that, the probability of $(||A\vec{b}||_\infty - d/2)$ being less than or equal to $O(\sqrt{n\log n})$ is $(1 - 1/n)$. With the observed fact that $||A^*\vec{b}||_\infty \ge d/2$, we conclude that with probability greater than $(1 - 1/n)$, the difference between $||A\vec{b}||_\infty$ and $||A^*\vec{b}||_\infty$ is less than $O(\sqrt{n\log n})$. $\square$

In fact, we have a tighter bound of $||A\vec{b}||_\infty - ||A^*\vec{b}||_\infty$ if the connection graph is *dense*. Specifically, if $d \ge \sqrt{12\log n}$, the following equation holds:

$$Pr(||A\vec{b}||_\infty \ge (d/2 + O(\sqrt{d\log n})) \le 1/n$$

Induced by the Theorem 5, taking object IDs as the numbering (i.e., in Algorithm 3) would be guaranteed to produce a partition which is almost optimal.

7. EXPERIMENTAL STUDY

7.1 Implementation Issues

We use Apache Spark [2] as the experimental platform. Spark is one of the most popular MapReduce-like platform which uses in-memory cache to gain high speedup against Apache Hadoop []. Since Spark directly supports the MapReduce paradigm, our algorithm is able to be easily implemented in Spark. In order to help reproduce our experiments, we further address some implementation issues.

[1]Every element in $\vec{b}$ is 1

[2]http://spark.apache.org/

7.1.1 Task Assignment

In spark, each task in reduce phase is an Apriori mining phase of a star. Although our star partition method is theoretically balanced, it is still necessary to assign equal number of tasks to each executors. Spark naturally uses hashing to partition data into tasks, where such a partitioning does not care on the tasks size. In order to fully utilize the clusters, it is important to perform a weight-aware partition. In our implementation, we collect the number of edges in each star after map phase. Afterwards, we use a simple *best-fit* strategy, where we assign stars in decreasing order with their sizes and each star is assigned to the currently least-loaded executor. HERE WE MAY HAVE ANOTHER BOUND FROM LITERATURE, BUT I DIDN'T FIND YET. The injection of load balancing strategy between map and reduce phase can be naturally implemented in Spark, where the map result can be cached and the reduce phase can be paused until when the partition strategy is ready.

7.1.2 Duplication Detection

It is notable that the patterns discovered from different tasks (stars) could be redundant due to containment relationship. For example, a pattern $\{a, b, c\}$ can be discovered from the star Sr_a, while the pattern $\{b, c\}$ can be discovered from the star Sr_b. Though in most applications, such a duplicate pattern is permitted, we offer an option to eliminate these patterns. The strategy is to broadcast each reducers output to every other reducers. This can be efficiently done via *broadcast* variables [3] in Spark. Afterwards, each reduce can check whether any resulted patterns are subsumed and thus filter those patterns. Theoretically, advanced techniques, such as Bloom Filters, can be applied to efficiently deal with the duplication detection. However, as the number of final patterns are normally quit small, we leave the exploration for those techniques to the future.

7.1.3 Handling Overlapping Clusters

When handling patterns such as *flock* and *group*, disk-based clustering on objects are applied. Such a clustering method may result one object belonging to multiple clusters. In such a case, just keeping the timestamps in the edge of connection graph is insufficient. Instead, we extend every timestemp t to a pair $\langle t, C\rangle$, where C is the set of clusters objects belong to at time t. The only adaption we need to take the join during apriori phase. Given two timestamp set T_1 and T_2, the join result of T_1 and T_2 instead of being $\{\forall t | t \in T_1 \wedge t_{\in} T_2\}$, it changes to $\{\forall (t, C) | t \in T_1 \wedge t \in T_2 \wedge C = (T_1.C \cap T_2.C) \wedge C \neq \emptyset\}$. It is obvious to see the *edge simplification* and *candidate pruning* still holds under this new setting.

7.2 Experimental Setup

Our experiments run on a 9-node cluster, with Apache Yarn as the cluster manager. We use 1 node for Yarn resource manager, and use the remaining 8 nodes as executors in Spark. In the cluster, each node is uniformly equipped with a 2.2GHz quad-core CPU with 32 GB memory. Inter-node communication is carried by the 1Gbps Ethernet. Some critical configuration of Spark is as follows:

[3]http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables

Parameter	Value
Java Version	1.7.0
spark.driver.memory	2GB
spark.executor.cores	2
spark.executor.instances	11
spark.executor.memory	7GB
spark.master	yarn-cluster
spark.serializer	KryoSerializer

We setup HDFS on the same cluster and all trajectory data are initially stored in the HDFS. We prepare two set of large-scaled trajectories with the following properties:

8. CONCLUSION AND FUTURE WORK

9. REFERENCES

[1] Y. Zheng, Y. Liu, J. Yuan, and X. Xie, "Urban computing with taxicabs," in *Proceedings of the 13th international conference on Ubiquitous computing*, pp. 89–98, ACM, 2011.

[2] Z. Li, B. Ding, J. Han, R. Kays, and P. Nye, "Mining periodic behaviors for moving objects," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1099–1108, ACM, 2010.

[3] J. Bao, Y. Zheng, D. Wilkie, and M. F. Mokbel, "A survey on recommendations in locationbased social networks. submitted to," *Geoinformatica*, 2013.

[4] X. Li, *Managing moving objects and their trajectories.* PhD thesis, National University of Singapore, 2013.

[5] J. Gudmundsson and M. van Kreveld, "Computing longest duration flocks in trajectory data," in *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pp. 35–42, ACM, 2006.

[6] Y. Wang, E.-P. Lim, and S.-Y. Hwang, "Efficient mining of group patterns from user movement data," *Data & Knowledge Engineering*, vol. 57, no. 3, pp. 240–282, 2006.

[7] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, "Discovery of convoys in trajectory databases," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1068–1080, 2008.

[8] Z. Li, B. Ding, J. Han, and R. Kays, "Swarm: Mining relaxed temporal moving object clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 723–734, 2010.

[9] Y. Li, J. Bailey, and L. Kulik, "Efficient mining of platoon patterns in trajectory databases," *Data & Knowledge Engineering*, 2015.

[10] J. Gudmundsson, M. van Kreveld, and B. Speckmann, "Efficient detection of motion patterns in spatio-temporal data sets," in *Proceedings of the 12th annual ACM international workshop on Geographic information systems*, pp. 250–257, ACM, 2004.

[11] Y. Zheng, "Trajectory data mining: an overview," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 3, p. 29, 2015.

[12] D. Birant and A. Kut, "St-dbscan: An algorithm for clustering spatial–temporal data," *Data & Knowledge Engineering*, vol. 60, no. 1, pp. 208–221, 2007.

[13] P. Laube, M. van Kreveld, and S. Imfeld, "Finding remodetecting relative motion patterns in geospatial lifelines," in *Developments in spatial data handling*, pp. 201–215, Springer, 2005.

[14] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112–122, 1973. doi:10.3138/FM57-6770-U75U-7727.

[15] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.